



# Test Driven Development

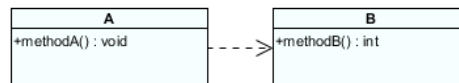
## Unit Testing with Mock Objects



1

## Dependencies: wat?

- Als klasse A in een methode een methode oproept van klasse B, heeft klasse A **een dependency op** klasse B



- In code:

```

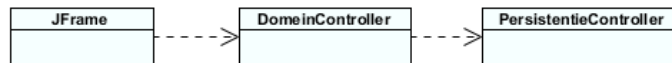
public class A {
    public void methodA() {
        B b = new B();
        int i = b.methodB();
    }
}

public class B {
    public int methodB() {
        return 666;
    }
}
  
```



## Dependencies: probleem

- In business applicaties hebben veel klassen dependencies:



- **Problemen bij het unit testen:**

– Als je een unit test doet van klasse A, voer je ook code van de klasse B uit.

Je test dus niet één klasse, maar meerdere klassen.

– Als de code van klasse B traag is, is het unit testen van klasse A traag

## Dependencies: probleem

- **Problemen bij het unit testen:**

– Als klasse B nog niet geschreven is (door een andere teamlid), kan je klasse A niet unit testen

– Als klasse B een user interface is, kan je klasse A niet unit testen zonder interactie met de gebruiker.

– Het is soms moeilijk het gedrag van klasse A te unit testen in het geval dat klasse B exceptions werpt.

– Testen met de database zijn vaak te traag.



## Oplossing Mock object

Een mockobject is een (software)object speciaal gemaakt om de eigenschappen en gedragingen te simuleren van een of meerdere objecten tijdens een softwaretest, zoals een unittest.

Een mockobject is een testtool.

Een mockobject is te vergelijken met een crashtest-dummy bij het testen van auto's, of een stuntman.



## Mock object

→ Mock objecten in de context van unit tests zijn 'nep'-objecten die het gedrag van echte objecten simuleren wanneer een echt object (of diens gewenst gedrag) niet makkelijk beschikbaar is.

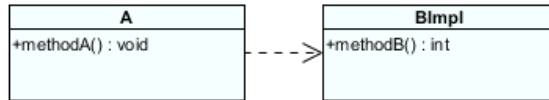
De gemockte objecten worden gebruikt door echte code die getest wordt.

Het mock object wordt door "**Dependency Injection**" in de echte code geplaatst.

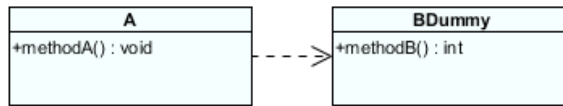
## Oplossing: Mock object



- Bij de echte uitvoering moet klasse A de echte klasse B gebruiken.



- Bij het unit testen van klasse A zou klasse A een dummy klasse moeten gebruiken die dezelfde methoden bevat als klasse B, maar in deze methoden zo weinig mogelijk doet



Mock object



## Dependency injection



**Dependency injection** is een geavanceerd ontwerppatroon uit de Informatica.

Met 'dependency injection' is het mogelijk om klassen *losjes* te koppelen.

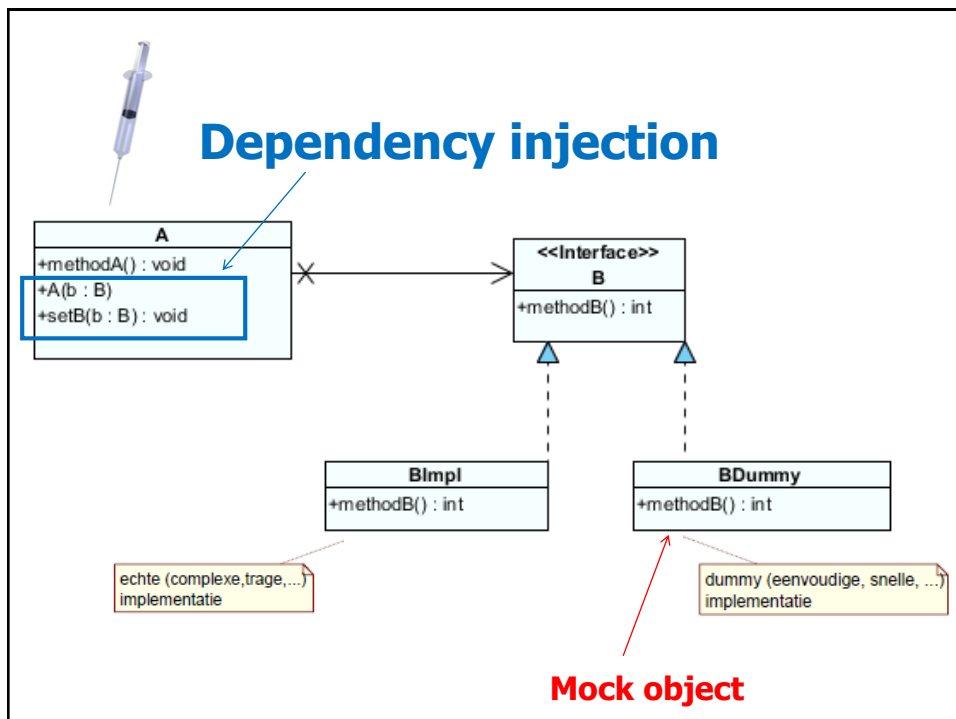
Dit wil zeggen dat ze data kunnen uitwisselen zonder deze relatie *hard* in de broncode vast te leggen; althans niet door de programmeurs van die (beide) klassen.

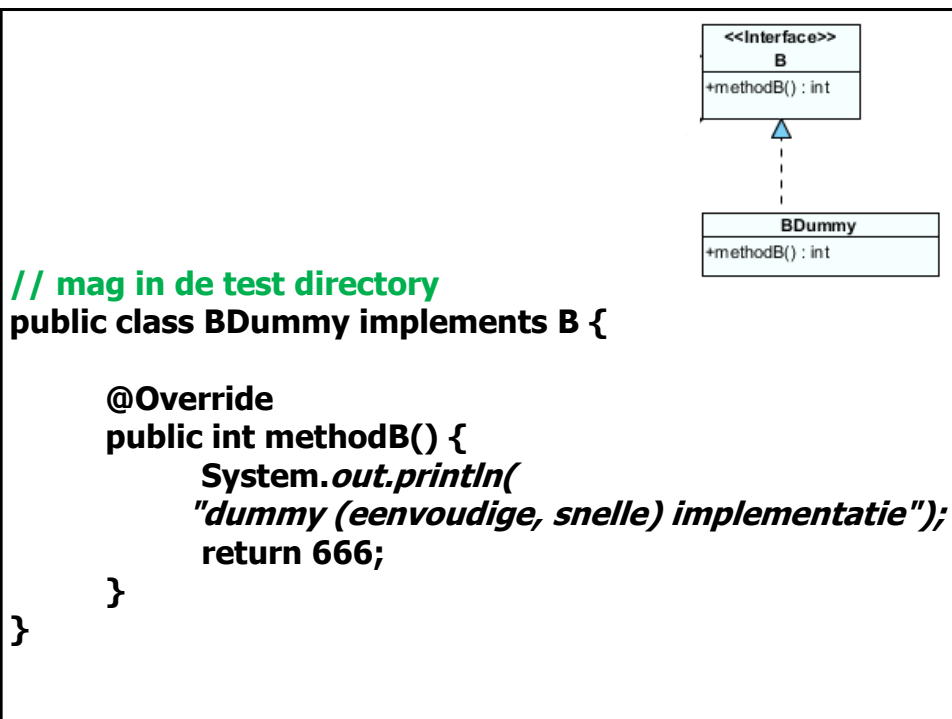
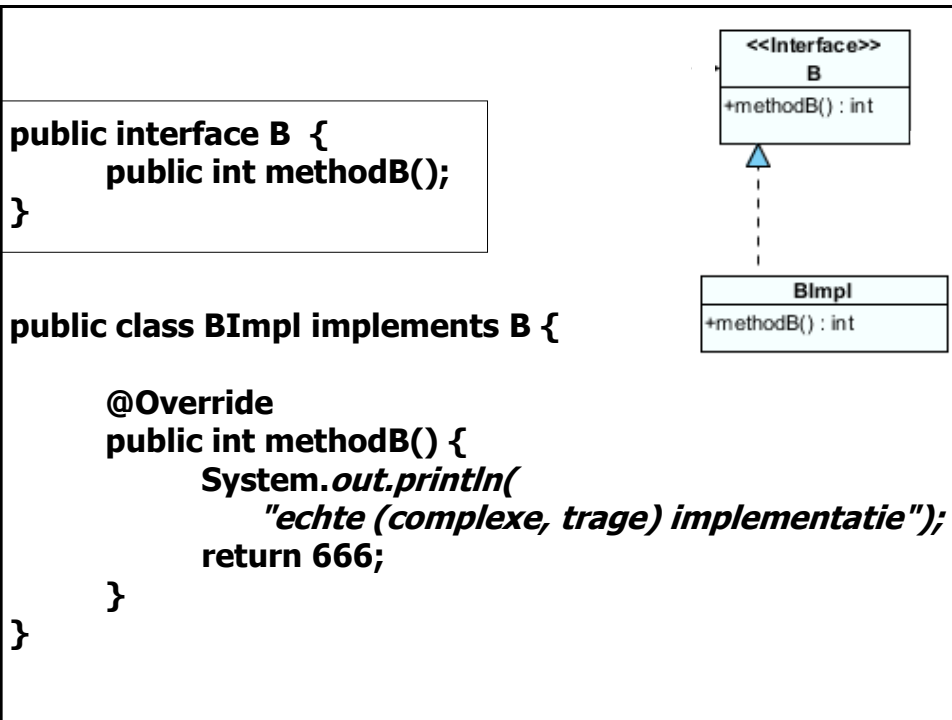
Traditioneel gebeurt dat vaak wel; waardoor die klassen moeilijk te hergebruiken zijn.

# Dependency injection



- BImpl en BDummy implementeren eenzelfde interface B
- Je drukt de dependency in A uit met een private variabele v.h. type B
- Je maakt in klasse A
  - Een constructor met een parameter van het type B
  - Een set methode met een parameter van het type B
- Bij het echt uitvoeren geef je A een object BImpl
- Bij het testen geef je A een object BDummy





OOAD

```

public class A {

    private B b; // dependency voorgesteld als interface type

    public A() {
        // echte uitvoering van A
        this(new BImpl());
    }

    public A(B b) { // constructor injection
        // b wordt vanuit Unit test geïnjecteerd met
        // Bdummy instance
        this.b = b;
    }
}

```

A


-b : B

+methodA() : void

+A(b : B)  
 +setB(b : B) : void

OF

+A()



OF

OOAD

```

public class A {

    private B b; // dependency voorgesteld als interface type

    public A() {
        // echte uitvoering van A
        setB(new BImpl());
    }

    public void setB(B b) { // setter injection
        // b wordt vanuit Unit test geïnjecteerd met
        // BDummy instance
        this.b = b;
    }
}

```

A


-b : B

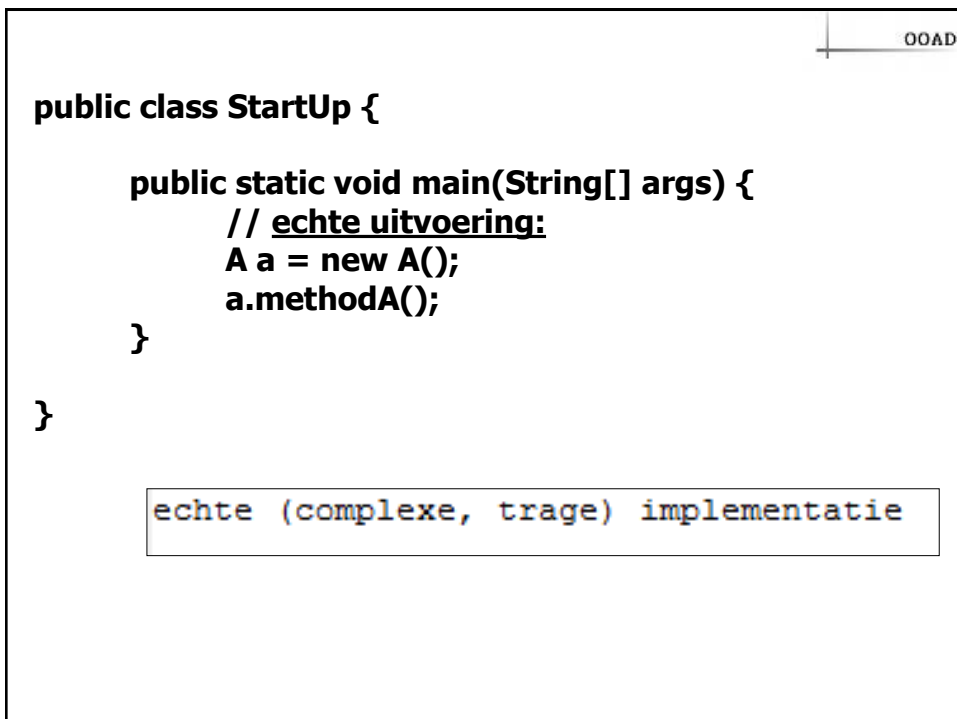
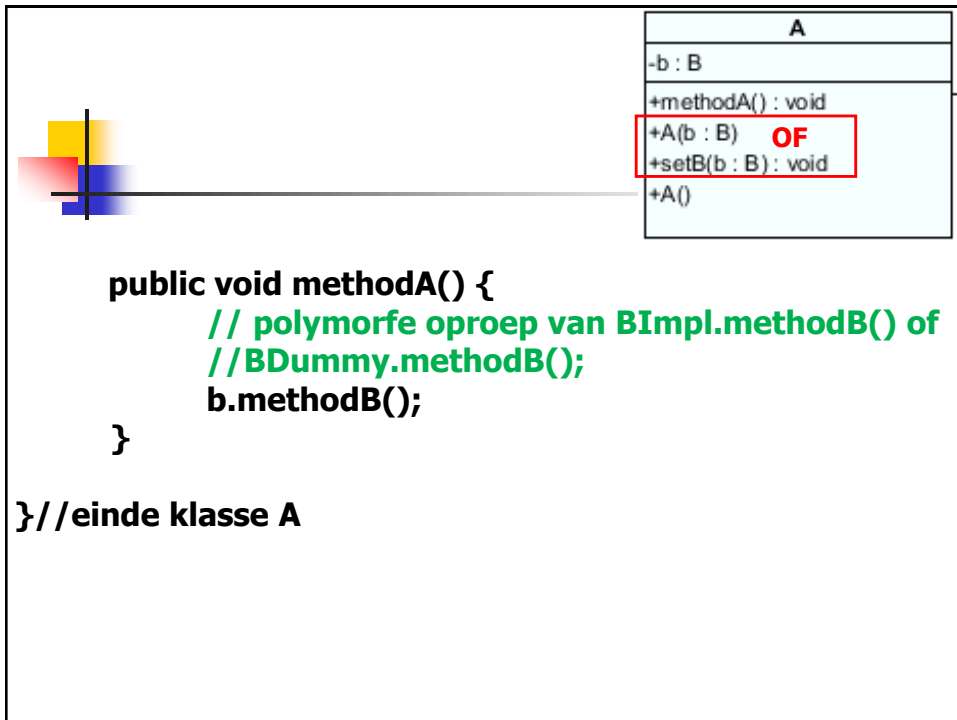
+methodA() : void

+A(b : B)  
 +setB(b : B) : void

OF

+A()







OOAD


```

import org.junit.Before;
import org.junit.Test;
public class ATest {
    private A a;

    @Before
    public void before() {
        // injecteer a met dummy implementatie:
        a = new A(new BDummy());
    }

    @Test
    public void testMethodA() {
        a.methodA();
    }
}

```



dummy (eenvoudige, snelle) implementatie

testen.ATest at
 
 100,00 %

The test passed (0.14 s)



## Design pattern – Inversion of control and Dependency injection

Soms wordt '*Dependency injection*' gezien als een bijzondere vorm van '*inversion of control*'.

### Inversion of Control:

"Inversion of control (IoC) is an abstract principle describing an aspect of some software architecture designs in which the flow of control of a system is inverted in comparison to procedural programming."



Also known as the **Hollywood Principle**  
(Don't call me, I'll call you)

Database: table landen:

code	oppervlakte
B	30528
NL	41528
LU	2586

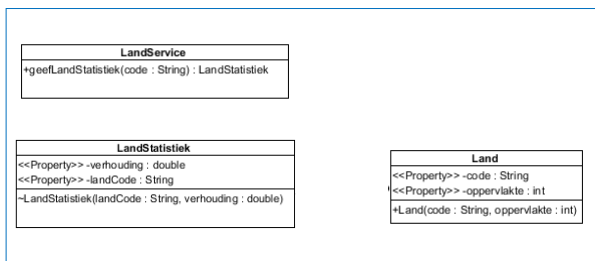
## Oefening

OOAD

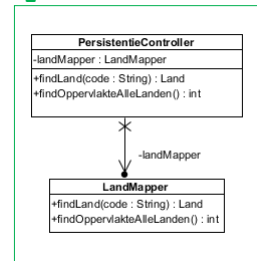
Geef van één land de code, oppervlakte en verhouding t.o.v. de totale oppervlakte van alle landen.

Klassen:

domein



persistentie



Gevraagd: Schrijf een test voor de methode "geefLandStatistiek(code: String): LandStatistiek"

package domein;  
public class Land {

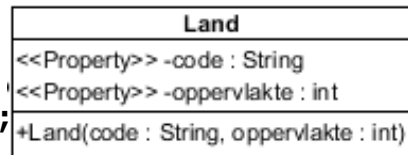
private final String code;  
private final int oppervlakte;

public Land(String code, int oppervlakte) {  
    this.code = code;  
    this.oppervlakte = oppervlakte;  
}

public String getCode() {  
    return code;  
}

public int getOppervlakte() {  
    return oppervlakte;  
}

}



## Oefening

```
package domein;
public class LandStatistiek
```

```
{
    private final String landCode;
    private final double verhouding;

    LandStatistiek(String landCode, double verhouding)
    {
        this.landCode = landCode;
        this.verhouding = verhouding;
    }

    public String getLandCode() {
        return landCode;
    }

    public double getVerhouding() {
        return verhouding;
    }
}
```

LandStatistiek
<<Property>> -verhouding : double
<<Property>> -landCode : String
~LandStatistiek(landCode : String, verhouding : double)

```
package domein;
import persistentie.PersistentieController;
```

```
public class LandService {
    public LandStatistiek geefLandStatistiek(String code) {
        if (code == null || code.trim().isEmpty()) {
            throw new IllegalArgumentException(
                "code mag niet leeg zijn");
        }
        PersistentieController persistentieController =
            new PersistentieController();
        Land land = persistentieController.findLand(code);
        if (land == null) {
            return null;
        }

        int oppervlakteAlleLanden = persistentieController
            .findOppervlakteAlleLanden();
        double verhouding = (double) (land.getOppervlakte())
            / oppervlakteAlleLanden;
        return new LandStatistiek(code, verhouding);
    }
}
```

LandService
+geefLandStatistiek(code : String) : LandStatistiek

```
package persistentie;
import domein.Land;
```

```
public class PersistentieController {
```

```
    private LandMapper landMapper;
```

```
    public Land findLand(String code) {
        if (landMapper == null) {
            landMapper = new LandMapper();
        }
        return landMapper.findLand(code);
    }
```

```
    public int findOppervlakteAlleLanden() {
        if (landMapper == null) {
            landMapper = new LandMapper();
        }
        return landMapper.findOppervlakteAlleLanden();
    }
```

```
}
```

PersistentieController
-landMapper : LandMapper
+findLand(code : String) : Land
+findOppervlakteAlleLanden() : int



## Besluit Mock objecten

- Mock objecten zijn *nep-objecten*. Ze worden gebruikt door echte code die getest wordt.
- Schrijf **geen 'business logic'** in mock objecten!

**"Some people used to say that unit tests should be totally transparent to your code under test, and that you should *not change runtime code to order to simplify testing*.**

**This is wrong!**

***Unit tests are first-class users* of the runtime code and deserve the same consideration as any other user.**

**If your code is too inflexible for the tests to use, **then you should correct the code**".**

